Optimizing Time and Energy in MPI Programs on a Power-Scalable Cluster

by

Robert C. Springer IV

(Under the direction of David K. Lowenthal)

Abstract

Recently, the idea that power is a performance-limiting factor has gained traction in the high-performance computing community. This may be due to the fact that the cost of energy has become increasingly significant, or that the heat produced by higher-energy components tends to reduce their reliability. In addition, cooling and providing ample power to supercomputers is becoming more difficult as their power requirements increase. One way to reduce power (and therefore energy) requirements is to use high-performance cluster nodes that are frequency- and voltage-scalable (e.g., AMD-64 processors).

The problem we address in this thesis is: given a power-scalable cluster and an upper limit for energy consumption, choose a schedule (number of nodes and CPU speed per node) that simultaneously (1) satisfies an external upper limit for energy consumption and (2) minimizes execution time. We do this using a novel combination of modeling, execution and profiling. Using our technique, we are able to find a near-optimal schedule in just a handful of partial program executions.

Index words: MPI, power scalable, clusters, profiling
Optimizing Time and Energy in MPI Programs on a Power-Scalable Cluster

by

Robert C. Springer IV

B.S., Emory University, 2002

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree Master of Science

Athens, Georgia

2005
Optimizing Time and Energy in MPI Programs
on a Power-Scalable Cluster

by

Robert C. Springer IV

Approved:

Major Professor: David K. Lowenthal

Committee: Shelby Funk
Kang Li

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2005
Acknowledgements

In completing this work, I must express my gratitude to my advisor, Dr. David Lowenthal, without whom I would never have gained so much from my experience at UGA. Dr. Lowenthal epitomizes the good professor: enthusiastic about teaching, deeply concerned with his students’ progress, eager to advance research in his field, and able to relate to his students as more than just a professor. This work owes a great deal to Dr. Lowenthal, both in direction and in being the focus of his motivation. I suppose the strongest endorsement that I can give is to say that, were I to do this all over again, I would still ask Dave to be my advisor.

I also thank Drs. Funk and Li for being on my committee; addressing the questions during my defense greatly strengthened my work and made this a much more complete investigation. I also thank them for their flexibility, as coordinating a defense from a thousand miles away had potential to be difficult. Fortunately, with such a committee, I experienced no such problems.

In addition, I would like to thank the faculty and staff of our wonderfully supportive Computer Science department. Whenever I had an odd request or needed last-minute support, I was always able to depend on those in the department here.

Finally, I must thank Meaghan Andrews, my companion for my two years in Athens. Attending UGA has been a wholly positive experience, and much of that comes from the fact that I’ve spent my time here with this sublimely wonderful woman. Meaghan, I love you with all my heart, and will never be able to fully repay the happiness you’ve given me during our time here.

However, in the upcoming years, I hope to try.
# Table of Contents

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
</tbody>
</table>

## Chapter

1. **Introduction**                        | 1
2. **Related Work**                      | 4
   2.1 Performance Prediction and Tuning | 4
   2.2 Energy-Related Research           | 5
3. **Implementation**                    | 7
   3.1 Assumptions                       | 8
   3.2 Model                             | 8
   3.3 Algorithm                         | 10
   3.4 Implementation Details           | 15
4. **Performance**                       | 17
   4.1 NAS Parallel Benchmarks          | 17
   4.2 Detailed Results                 | 20
5. **Conclusion**                        | 27

Bibliography                            | 28
List of Figures

4.1 Comparison of predictions vs. actual executions for LU. . . . . . . . . . . . . 21
4.2 Categories of speedup and OPM. . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.3 Effect of the energy limit on executions required . . . . . . . . . . . . . . . . . 25
LIST OF TABLES

3.1 Terms used in model .......................................................... 9
3.2 Illustration of algorithm on SP with energy limit of 37 kJ .............. 13
4.1 Results of running our algorithm on the NAS parallel benchmarks .... 18
4.2 Average difference between predicted and actual results .............. 22
4.3 Results of running algorithm on synthetic benchmarks .............. 24
4.4 Performance of CG with and without superlinearity filtering ........ 26
Recently, power-aware computing has gained traction in the high-performance computing (HPC) community. As a result, low-power, high-performance clusters, such as Blue-Gene/L [1], have been developed to satisfy the ever-increasing demand for energy, while maintaining good performance. Blue Gene/L and similar systems solve this problem by utilizing low-power processors. However, since the processors are only designed to use one voltage, the system uses the same amount of energy under load, regardless of the characteristics of that load. Our previous work [17] has shown this can be inefficient for some applications.

Our research focuses on using small- to medium-size power-scalable clusters, which utilize processors that are each frequency and voltage scalable—i.e., their clock speed and therefore power consumption can be changed dynamically. Such clusters can potentially deliver good energy efficiency because an increase in CPU frequency generally results in a smaller increase in application performance. The reason for this is that the CPU is frequently not the bottleneck resource.

However, while the CPU frequency of a node (in our case, a node is a single processor) can be scaled down to save energy, such scaling saves only CPU energy. A much greater savings is possible by simply not using a node; i.e., a given program can be run on only a subset of the available nodes. Hence, given a power-scalable cluster, there are two primary ways to save energy: (1) power down a subset of the nodes, and (2) on the nodes that are actively participating in the computation, scale down the CPU frequency. This thesis synthesizes these two approaches.
The problem we address in this thesis is: given a target program, a power-scalable cluster and an upper limit for energy consumption, choose a schedule that simultaneously satisfies the energy limit and minimizes execution time. In this paper the term schedule is used to mean a tuple whose first element specifies the number of nodes to be used; the remaining elements indicate, for each computational phase, the CPU speed in which to execute that phase. Because determining the optimal schedule requires (in the worst case) executing all possible schedules, the problem is exponential in terms of the number of phases. Hence, the goal of this paper is to determine a near-optimal schedule, while only executing a few iterations of a given program a small number of times.

The approach we propose uses a combination of performance prediction and profiling, backed up by actual program execution. First, we create models for both execution time and energy consumption. Second, we create initial estimates of the key parameters of the model by executing a small number of iterations and subsequently performing regressions. After these two steps, we repeatedly execute the following steps until a “good” schedule is found: (1) use the model to create predictions for each possible schedule, (2) execute a few iterations of the program using the best predicted schedule, and (3) update the estimates of the key parameters using the additional information collected during the executed iterations.

For evaluation, we used a combination of the NAS parallel benchmarks and several synthetic benchmarks. This allows us to give both a real-world evaluation and an exploration of the bounds of the problem space. Using our technique, we are able to find a near-optimal solution in just a handful of partial program executions. For example, for the NAS FT benchmark, by using 14 executions of a few iterations, we discovered a schedule that results in a completion time that is within 1% of the optimal. An exhaustive search would have required execution of over 100 schedules. This result is made possible because our model of execution time and energy consumption typically results in predictions that are within 10% of the measured values. In addition, the results are promising not just for FT, but also for the
other NAS benchmarks, as well as a set of synthetic programs that cover a range of different application characteristics in terms of parallel speedup and exerted memory pressure.

The rest of this thesis is organized as follows. Section 2 describes related work. Section 3 discusses our model and algorithm for finding an effective schedule. Section 4 discusses the measured results on our power-scalable cluster. Finally, Section 5 summarizes and describes future work.
In this section we describe some of the closely related research. We divide the related work into two broad categories: (1) performance prediction and tuning and (2) energy-related research in both server/desktop and mobile systems.

2.1 Performance Prediction and Tuning

First, underlying this work is the problem of understanding parallel scalability. The work in [40, 41] focuses on finding MPI operations that cause scalability problems. This is done though both machine learning and statistical analysis. In [38], a model for understanding the scalability of a class of task and data parallel programs is presented.

Second, several researchers have solved various parallel programming-related problems by (1) executing programs, (2) taking measurements, and then (3) analyzing the results. Perhaps the best known of these is ATLAS [45], which uses the Automated Empirical Optimization of Software technique. Essentially, a specialized library of linear algebra functions is created by executing the library functions over several days, with many different compile-time options. The ADAPT system took a similar approach [42]. The ATLAS technique was generalized to other high-performance computing kernels in [46]. Other related techniques include executing a few iterations of a high-performance application to predict performance across different platforms [47] and executing MPI routines on each new platform to generate the most efficient implementation of each [12].

This thesis borrows several of the general ideas given above, but differs in several ways. First, the number of executions that our technique can make is limited, because we are not
creating a library, but rather running a program that the user wishes to complete as soon as possible. Second, we address the need to understand program scalability by running a (small) set of iterations of the program and developing time and energy models that both encompass computation and communication.

2.2 Energy-Related Research

Several researchers have investigated saving energy in server-class systems. In sites such as hosting centers where there is a sufficiently large number of machines, energy management may become an issue; see [5, 33, 32, 11] for examples of this using commercial workloads and web servers. Such work shows that power and energy management are critical for commercial workloads, especially web servers [27]. Additional approaches have been taken to include dynamic voltage scaling (DVS) and request batching [10]. The work in [36] applies real-time techniques to web servers in order to conserve energy while maintaining quality of service.

Our project differs from most prior research because it focuses on HPC applications and installations, rather than commercial ones. A commercial installation tries to reduce cost while servicing client requests. On the other hand, an HPC installation exists to speed up an application, which is often highly regular and predictable. One approach is to save energy in an application-specific way; the work in [6] used this approach for a parallel sparse matrix application. Another HPC effort that addresses the memory bottleneck is given in [21]; however, this is a purely static approach. This paper also introduced the concept of $\beta$; also, $\beta$ was adapted for HPC in [22]. Also, Cameron et al. [13] developed a measurement infrastructure for power-aware HPC programs on a cluster of laptops and performed experiments on NAS programs. Finally, metrics for power-aware HPC are described in [23]. None of this work, however, considers finding a schedule.

In server farms, disk energy consumption is also significant; several have studied reducing disk energy [4, 51, 19, 20, 31]. In this thesis we do not consider disk energy, as it is relatively minor compared to CPU energy.
There are also a few high-performance computing clusters designed with energy in mind. One is BlueGene/L [1], which uses a “system on a chip” to reduce energy. Another is Green Destiny [43], which uses low-power Transmeta nodes. A related approach is the Orion Multi-system machines [30], though these are targeted at desktop users. Unlike our approach, these machines sacrifice performance in order to save energy by using less powerful processors.

While not our focus in this work, there is also a large body of work in saving energy in mobile systems. At the system level, there is work in trying to make the OS energy-aware through making energy a first-class resource [39, 9, 7]. One important avenue of application-level research on mobile devices focuses on collaboration with the OS [28, 44, 48, 49, 50, 15, 14, 2]. Our approach differs in that we are concerned with saving energy in HPC applications, where execution time is the primary consideration.

Our prior work is threefold: (1) an evaluation-based study that focused on exploring the energy/time tradeoff in the NAS suite [17], (2) development of an algorithm for switching CPU frequency, or gear dynamically between phases [16], and (3) leveraging load imbalance to save energy [25].

In [16], we establish the usefulness of varying the gear per phase and providing an algorithm for choosing the assignment of gear to phase. This thesis extends this idea by allowing the number of nodes to vary, which complicates the problem considerably.

Finally, in our work we divide programs into one or more computational phases. There has been a large body of work in phase partitioning. Static techniques, such as [26, 35, 29], appeared in the literature first. More recently, dynamic techniques have been used [24, 8, 37].
Chapter 3

Implementation

This section describes our implementation. The inputs to our system are a program divided into P phases, an energy limit, and a maximum number of available nodes. Our system will output a \((P + 1)\)-tuple schedule, the first element of which is the number of nodes to use, and the remainder the gear selected for each phase. To determine a schedule, we use a combination of performance prediction and profiling based on a small number of program runs of only a few iterations each.

Our system takes as input a program divided into one or more phases. For this thesis, we used a straightforward phase division technique, which applies primarily to iterative and predictable HPC applications. We first obtain a trace of the application in question (using the fastest gear on each node, i.e., the highest frequency-voltage setting). The division into phases is done by examining the trace and using an ad hoc approach that conforms to the following heuristics. First, any MPI operation demarcates a block boundary. Second, if the memory pressure changes abruptly, a block boundary occurs at this point. (Memory pressure is determined by inspecting performance counters; for this paper we look at operations and cache misses.) We have found in previous work that multiple phases are necessary to obtain good time/energy tradeoffs in several of the NAS benchmarks. For example, in MG, by using multiple phases we were able to save an additional 11% energy compared to using a single phase (and therefore only a single gear). Further details are available in [16].

This section first describes our assumptions and then presents our time and energy models. Next, we give our algorithm for finding an effective schedule. Finally, we discuss specific details of our algorithm.
3.1 Assumptions

First, the effectiveness of our system relies on having effective partitionings of phases. We consider the general phase division problem to be orthogonal to our research, but can leverage off of a large amount of related work in the area (described in Section 2).

Second, we assume that we are given an maximum (total) energy constraint (by the user or cluster administrator). While not currently policy on large supercomputers, we can envision a situation in the near future where the cost of energy is charged, at least partially, to the user. This constraint may be imposed for many reasons; for example, the aggregate mean time to failure of large clusters has fallen into the range of minutes [23]. The Arrenhius equation states that a $10^3K$ reduction in operating temperature will result in a doubling of expected hardware lifetime [34]. Because energy consumed translates directly into heat dissipated, reducing energy consumption should extend hardware lifetimes.

Third, we assume no load imbalance. In future work we will address load imbalance, but the programs in the NAS suite generally balance the computation between nodes fairly evenly. Extending the model described in the next subsection to handle load imbalance is a nontrivial task. Finally, we assume that program behavior between iterations is consistent, so that it is possible to predict future program behavior by examining current behavior.

3.2 Model

In this section we describe our models of execution time and consumed energy. Later, we discuss the algorithm we use to predict time and energy in as few iterations as possible. The variables used in our model are summarized in Table 3.1.

In our previous work [17], we noted that (1) computation time and idle time (which is due to communication) tend to scale differently as the number of nodes increases, and (2) the power consumed by the system is different when computing than when idle. There-
<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_p$ ($F_s$)</td>
<td>Percentage of code that is parallelizable (sequential)</td>
</tr>
<tr>
<td>$T^C$</td>
<td>Time spent in computation</td>
</tr>
<tr>
<td>$T^B$</td>
<td>Time spent in blocking communication</td>
</tr>
<tr>
<td>$T_g$</td>
<td>Time spent in a region when executing at gear $g$</td>
</tr>
<tr>
<td>$T^C_g$</td>
<td>Computation time if executing at gear $g$</td>
</tr>
<tr>
<td>$S_g$</td>
<td>Computation slowdown factor if executing at gear $g$</td>
</tr>
<tr>
<td>$P^C_g$</td>
<td>System power during computation at gear $g$</td>
</tr>
<tr>
<td>$P^B_g$</td>
<td>System power when idle at gear $g$</td>
</tr>
</tbody>
</table>

Table 3.1: Terms used in model

Therefore, our model separates execution time into its computation and blocking communication components.

By using Amdahl’s law, the computation component of a program can be decomposed into parallelizable and inherently serial fractions ($F_p$ and $F_s$, respectively). For each different number of nodes, these fractions will be different. To compute these fractions on $n$ nodes, the equations are:

$$T^C(n) = T^C(1)(F_p/n + F_s)$$

$$F_p = 1 - F_s$$

Given $F_p$ and $F_s$ for a given number of nodes, the equations above can be used to generate computation time.

If we wish to predict execution time for a gear other than top gear, we need to consider the effect of gear slowdown, $S_g$. First, we assume that any time spent or power consumed in blocking communication calls is unaffected by gear because the CPU is mostly idle (i.e., $T^B$ and $P^B_g$ are constants). We focus now on the effect of a gear change on computation time.
Because changing gears implies a change in clock frequency, the computation time is altered by a gear change. If we know the slowdown $S_g$ in shifting from top gear (gear 0) to a slower gear $g$ ($g > 0$), then the new computation time is $T_g^C = T_0^C \times S_g$. Therefore, the equation for $T_g$ is: $T_g = T_0^C + T^B$. Once $T_g$ is known, the energy consumption can be calculated by the formula $E_g = P_g^C T_g^C + P_g^B T^B$.

While these equations are relatively straightforward, there are two situations where they are not fully precise. First, we assumed that $T^B$ and $P_g^B$ are constants. It turns out that they are actually affected by the gear. We further discuss how to extend the model to improve its accuracy in Section 3.4.

Second, in our earlier work [17], we discussed the effects of reducible time, on execution time when changing gears. Reducible time is defined as the time between an send/receive or send/wait pair. It is important because when using a lower gears, computation speed slows but communication speed remains relatively constant. Any reducible work increase, therefore, will result in a corresponding idle time ($T^B$) decrease (unless $T^B$ reaches zero).

We analyzed program traces to determine the amount of reducible time for the programs considered for these experiments and found that all but one of them had essentially no reducible time\(^1\). Thus, we elected not to consider the effects of reducible time in this thesis, but we will consider this in future work.

### 3.3 Algorithm

Broadly speaking, our algorithm repeatedly executes the program for a handful of iterations. On each run, we measure time and energy as well as refine our estimates for a subset of the dependent variables ($F_p$, $F_s$, $S_g$, and $P_g^C$) in our models. This improves our overall estimate of execution time and energy consumption. This section provides details of our algorithm.

\(^1\)The NAS SP benchmark had 6s of reducible time out of 60s of computation time. Further inspection revealed that any prediction inaccuracy due to reducible time was at most 1s.
Step 1: Initialization. The first step is to determine initial values for $P_g^C$, $F_p$, $F_s$, and $S_g$. The minimum number of runs to do this is $g + 3$. The first $g$ runs are required to determine $P_g^C$ for each energy gear. This is done by executing the program on one node and dividing the observed energy consumption by the execution time. One node is used so that communication does not affect our measurements. In other words, we are only measuring computation power.

The remaining 3 runs are necessary to create accurate estimates for $T_C$, $T_B$, and $S_g$. For $T_C$, we use a regression on measured values for $F_p$ and $F_s$. Our previous work has shown that for programs without scaled speedup, $F_p$ and $F_s$ scale linearly with the number of nodes. Therefore, a linear regression using these values is an accurate way to predict their values at unknown numbers of nodes.

Communication time may scale in different ways depending on the implementation of a specific program. In most cases, communication time scales either logarithmically, linearly, or quadratically. Knowing which of these models best fits the communication of a particular program requires three runs. With the observed data, we fit the data to each of the three functions, and select the function that gives the lowest average error.

We found that the best way to measure $S_g$ was to use micro-ops per data cache miss, $OPM$. Our previous work has showed that OPM correlates well with memory pressure [18]. Through experimentation, we found that a logarithmic function best fits the relation between OPM and $S_g$. We use this to estimate $S_g$ for unknown numbers of nodes. The reason we use OPM to indirectly measure $S_g$ is because $S_g$ is not constant across numbers of nodes; indeed, cache performance (which is directly related to how much pressure is exerted on memory) often improves when nodes are added.

Step 2: Predict and sort all schedules. Because the time taken to predict execution time and consumed energy (as opposed to executing the program) for a given schedule is negligible, we make predictions for every possible schedule. We divide the schedules into two lists: those predicted to satisfy the energy constraint (denoted satisfying), and the remainder (denoted non-satisfying). Note that because the power meter carries with it a small error
(see Section 4.2 for details), a schedule is deemed satisfying if the energy consumed is smaller than the difference of the limit and the error.

Each list is then sorted: the satisfying list in order of increasing execution time, the non-satisfying list in order of increasing energy. In other words, the first schedule on the satisfying list is the one predicted to be fastest within the energy constraint, and the first schedule on the non-satisfying list is predicted to be closest to satisfying the energy constraint.

**Step 3: Execute schedule and update estimates.** We then select and execute (for a few iterations) the first schedule on the satisfying list. Note that in this step we do not ever select a test that was previously executed (see below). We next execute this schedule for a few iterations. The results are compared to the energy constraint, and if it indeed is below the constraint, we perform validation tests (step 4). Otherwise, we update our estimates of $F_p$, $F_s$, and $S_g$ as in step 1 and return to step 2.

**Step 4: Validate schedule.** Once we have found a schedule that satisfies the energy constraint, we wish to ensure that the proposed solution is close to the best possible schedule. To do this, we run two further tests: one which differs in the number of nodes, the other which differs in gear selection.

To ensure we are using the right number of nodes, we perform what we call node validation. We examine the non-satisfying list for schedules that use a greater number of nodes than the proposed solution\(^2\). Out of those schedules, we choose the one that is predicted to use the least energy that executes faster than the candidate schedule.

This schedule is then executed for a few iterations. If it does not satisfy the energy constraint, it is unlikely that any solution using more nodes will satisfy the energy constraint or execute in less time. If the test does satisfy the energy constraint, then we go to step 2 and regenerate new estimates for all schedules. Otherwise, we proceed to gear validation,

\(^2\)The NAS do not run on an arbitrary number of nodes, so the closest higher number of nodes may not be simply adding 1 to the current number.
which is similar in spirit to node validation: we consider a schedule using the same number of nodes, but with at least one phase using a faster gear. If this schedule does not satisfy the constraint, it means that any schedule using strictly faster gears and the same number of nodes will fail to satisfy the energy constraint.

At this point we have shown that there is likely no better legal schedule if the number of nodes or gears is changed, so our algorithm terminates and returns the candidate schedule. Presumably the user will now execute the entire program in this schedule.

On the other hand, if the gear validation test does satisfy the constraint, then, as before, it means the candidate solution may be able to be improved with better estimates, and so we go to step 2.

Note that it is possible that a validation test cannot find a valid schedule: this could occur if we are using the maximum number of nodes or top gear for all phases. In this case, that validation test is skipped. In addition, a given schedule is only run once during program execution. We store the results of every execution so that should a schedule be selected twice, the results of the first run can simply be re-used.
Example. We illustrate our algorithm for the specific case of SP with an energy limit of 37kJ. Table 3.3 shows the steps of our algorithm in detail. In the table, the entries in the Run column are one of the following: $I_1$, $I_2$, and $I_3$ are are the initialization tests, an $S$ denotes the accepted schedule, $G$ is a gear validation test, $N$ is a node validation test, and an integer denotes an ordinary test. If the actual energy value is below the limit, then that entry is in bold. A schedule is described by $(P, [g_0, g_1, \ldots])$, where $P$ is the number of nodes and $g_i$ is the gear to be used for phase $i$.

First, we run the initialization tests ($I_1$ through $I_3$). Then, we predict energy consumption and execution time for all schedules. Based on the predictions, we select $(4, [2, 1])$ as the fastest schedule predicted to satisfy the energy constraint. Then, we execute the program using this schedule for a few iterations and collect the energy consumed. Because it is below the energy limit, we attempt the gear validation test, which uses the schedule $(4, [1, 1])$. This is the schedule that uses the same number of nodes, but at least one faster gear, and is predicted to be faster than $(4, [2, 1])$ but to consume more energy than the limit. After execution of this test, we note that, contrary to our predictions, $(4, [1, 1])$ did satisfy the energy limit. Therefore we refine our estimates, return to step 2, and once again predict all schedules.

This time, the best predicted schedule is $(4, [1, 1])$ and, because it has already been evaluated, we begin gear validation again. This time the schedule $(4, [0, 1])$ is selected and executed. The result is that it does not satisfy the energy limit. Thus gear validation is successful, and we progress to node validation. For this, we select the schedule $(9, [4, 4])$. This is the schedule with more nodes that is predicted to be closest to, but not satisfying, the limit. Execution of this schedule showed that it did not satisfy the limit, so node validation is successful. This means that the algorithm terminates, returning $(4, [1, 1])$ as the selected schedule.
3.4 Implementation Details

This section describes some specific details of our implementation. First, we discuss how we collect data. Then we describe how we estimate $P_g^B$. Finally, we discuss the role of reducible computation.

Data Collection. To initialize the dependent variables ($F_p$, $F_s$, $S_g$, and $P_{gC}$) of the model, it is necessary to gather data from program executions. To do so, we utilize a combination of MPI call profiling, hardware performance counters, and inline wattmeters. As described above, we need to collect $T_C$, $T_B$, and OPM in order to estimate the above dependent variables.

To do this, we use our MPI-JACK tool, which intercepts MPI calls and allows for arbitrary code to be inserted before and after execution of the call. This was used to gather the duration of each MPI call as well as to inspect the performance counters to obtain micro-operations and cache misses (to determine OPM).

We determine energy consumption by placing a wattmeter between the system power supply and the wall socket. Power readings are taken from the wattmeter every second. Those values are then integrated over time to give an energy value.

Generalizing $P^B$. Earlier, we assumed that any time spent blocking on communication calls (e.g., MPI_Allgather and MPI_Recv) takes constant power. Separate results [18] have showed that the time is largely (but not completely) independent of gear. Therefore, we do not concern ourselves with generalizing $T^B$.

However, we found through experiments that the power consumed blocking on communication calls could be significantly more than idle power, and that it varied with gear. This is because there is a computation component of communication time that may be nontrivial. So, it is inaccurate to consider power due to communication for blocking calls to be identical to power consumed when the system is idle. Because different MPI calls have different
computation/idle ratios, we need to determine power due to communication separately for each call.

To address this issue, on any multiple-node run we use MPI-JACK to log all message events. We record the computation portion of communication routines as well as the energy consumed. $P_C^0$ is determined experimentally. Then, $P_B^g$ at top gear is determined by first using:

\[ P_B^0 = \frac{E/n - P_C^0 T_C}{T_B} \]

To calculate $P_B^g$ at all other gears, we need to know the percentage difference between CPU power consumption for all $g$. Because $P_C^g$ represents the combination of CPU and system power, it is necessary to record system-only power by taking readings when idle. Once system power has been measured, the CPU-only power is measured by subtracting the idle power from $P_C^g$. Then, $P_B^g$ can then be computed by multiplying $P_B^0$ by the CPU power at gear $g$ divided by CPU power at top gear.
This section reports our performance results. For all experiments, we use a 10 node AMD Athlon-64 cluster connected by a 100Mbps network. Each node has 1 GB of main memory. The Athlon-64 CPU supports energy gears of 2.0, 1.8, 1.6, 1.4, 1.2, 1.0, and 0.8 GHz, but the 1.0GHz gear was not reliable and so was not used. Each node runs the Fedora core 2 OS, and gear shifting was done through the \texttt{sysfs} interface. All applications were compiled with either \texttt{gcc} or the Intel Fortran compiler, using the \texttt{-O2} optimization flag. We controlled the entire cluster, so all experiments were run when the only other processes on the machines were daemons.

We first describe the overall results of running our system on the NAS suite. Then we present the results of more detailed examinations of the behavior of both our algorithm and model.

### 4.1 NAS Parallel Benchmarks

Table 4.1 displays the results of running our algorithm on all the NAS programs [3]. Note that for each program, we selected an energy limit as follows. First, we measured the energy consumed running at top gear for all phases, using 8 (or 9, for those benchmarks that require it) nodes. Then, we subtracted 15% from this amount of energy. (The limit shown is actually the original limit, minus a 1% error; the error is explained further in Section 4.2). The reason that we subtracted 15% was that this was enough in all cases to ensure that the optimal schedule was \textit{not} simply to use all nodes at top gear. It is important to note that if one selected a single constant energy limit for all NAS benchmarks, one of two things would
Table 4.1: Results of running our algorithm on NAS parallel benchmarks. The chosen energy limit is 15% less than the energy consumed at top gear for all phases, using 8 (or 9, for BT and SP) nodes. The time difference represents the difference in time when executing the program using our chosen schedule from that when executing the optimal schedule. Note that the “necessary” executions include eight executions required for initialization.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>61</td>
<td>11</td>
<td>108</td>
<td>(4, [0,0])</td>
<td>1</td>
</tr>
<tr>
<td>CG</td>
<td>20</td>
<td>12</td>
<td>24</td>
<td>(4, [1])</td>
<td>1</td>
</tr>
<tr>
<td>EP</td>
<td>43</td>
<td>11</td>
<td>24</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>FT</td>
<td>70</td>
<td>14</td>
<td>144</td>
<td>(8, [1,4])</td>
<td>7</td>
</tr>
<tr>
<td>IS</td>
<td>33</td>
<td>13</td>
<td>24</td>
<td>(8, [4])</td>
<td>1</td>
</tr>
<tr>
<td>LU</td>
<td>23</td>
<td>11</td>
<td>864</td>
<td>(8, [4,4])</td>
<td>1</td>
</tr>
<tr>
<td>MG</td>
<td>20</td>
<td>12</td>
<td>144</td>
<td>(8, [4,4])</td>
<td>2</td>
</tr>
<tr>
<td>SP</td>
<td>45</td>
<td>11</td>
<td>108</td>
<td>(9, [4,4])</td>
<td>3</td>
</tr>
</tbody>
</table>

have occurred: either the “trivial” solution of top gear/maximum nodes would be selected for many of the benchmarks, or many benchmarks would not be able to be run with any schedule because all schedules would have exceeded the limit. In the table, we show the number of executions incurred by our algorithm as well as the number of executions needed for an exhaustive search. In addition, we show the rank of the schedule (the $n^{th}$ best schedule has rank $n$) and most importantly, the time difference from optimal. The optimal solution was obtained through exhaustive search; of course, in general this is not feasible. Recall that each “execution” is not an execution of the entire program, but rather a small number (5 in the NAS programs) of iterations.

The table shows that we are able to produce an optimal or near-optimal schedule in all cases in which a schedule that satisfies the energy limit exists. Furthermore, in all cases the number of executions using our system was small. In particular, consider FT. In this case, our system found a schedule within 1% of optimal. Furthermore, this was the program in the NAS suite that required the largest number of executions—and it only required 14, 8 of
which are required initialization runs. Considering that running all schedule permutations requires 256 runs ($4 \times 6^2$, as there are four possible node configurations, six gears, and two phases), this is reasonable—and is the worst case over the entire NAS suite.

In fact, in many cases we achieved the minimum number of runs: 8 initialization runs and 3 subsequent runs (one prediction and then up to two validation tests\(^1\)). This was the case for BT, EP, LU, and MG. It is worth noting that the reason that FT required the most executions is because the energy constraint was such that many candidate schedules consumed close to that amount of energy. Based on the observed accuracy of the FT predictions, it is likely that fewer executions would be needed with a different constraint (we verified this with additional experiments). More details on the effect of the energy limit are given in Section 4.2.

We emphasize that the number of initialization runs remain constant as both the number of available nodes and the number of program phases increase—meaning that the initialization runs will be negligible on large clusters executing large-scale programs. Additionally, if the energy limit is changed, the initialization runs need not be repeated.

Next, we examine the quality of the chosen schedule. We deem our system to have chosen a high-quality schedule when it is one of the best possible schedules, and it results in an execution time that is at least close to that of the optimal schedule.

Table 4.1 shows that our system does indeed select a high-quality schedule in all cases. In fact, in all cases other than MG, FT, and SP, our algorithm selected the optimal schedule. For MG, the reason for the suboptimal schedule is most likely that the phase boundaries are not optimal. This is evidenced by the fact that the best 5 schedules at both 4 and 8 nodes are [0,0] through [4,4], where both phases use the same gear.

Changing gears involves a small but non-zero penalty in time, so a savings in energy may be offset by a larger increase in execution time, which would make the division of a program region into two subregions not beneficial. Therefore, we see that proper phase identification is important.

\(^1\)Fewer runs actually may occur in the case that one of the validation tests happens to be the same schedule as one of the initialization runs.
In the case of FT, the seventh-best schedule was chosen; yet, it is important to note that, as said above, this schedule produced an execution time that was within 1% of the optimal schedule. The reason we did not select the best schedule is discussed below; in short, it is due to the many schedules “near” the chosen schedule in the problem space. Finally, in the case of SP, we chose the schedule (9, [4,4]). Precision error caused our system to believe that (9, [2,2]) and (9, [3,3]) were above the energy limit, but they were not. Furthermore, they were not run during the verification test, because (9, [3,4]) was instead. Obviously, with more verification runs we could find the optimal schedule in this case, but that would also increase the number of executions in the common case.

Additionally, we note that for EP, no schedule satisfied the energy limit. For this program, 8 nodes at top gear consumes 50kJ, so the energy limit chosen was 43kJ. Because EP gets nearly perfect speedup, two things are true. First, 1, 2, and 4 nodes at top gear also consumes about 50kJ. Second, any schedule that uses a slower gear will not only take longer, but will also consume more energy. This is because EP is completely CPU bound, and for such programs, it is always advantageous to run them at the fastest gear.

Finally, we notice that many of the schedules happen to use gear 4 (1.2 GHz). We believe that this is because for the AMD-64, gear 4 happens to be an efficient gear. On the other hand, gear 5 (0.8 GHz) is, in our experience, an inefficient gear and is rarely chosen. Again, in only one case (SP) was gear 4 chosen when a different gear would have been better.

4.2 Detailed Results

In this section we examine in more detail several aspects of our model as well as the behavior of our algorithm. Specifically, we examine the accuracy of our predictions, behavior at the extremes of the problem space, sources of error, the effect of the choice of energy limit, and algorithm behavior given a program with a discontinuity in the speedup curve.
Accuracy of our model: In Table 4.1 we presented the performance of our algorithm combined with our model. The reason we are able to generate near-optimal schedules in few executions is because our model is accurate. To examine this further, we used our time and energy model to make predictions for all possible permutations of gears per phase (this totals 864 complete executions) on LU. Then we ran all programs, producing exhaustive results (as described above) to compare. The results are presented in Figure 4.1. Examining the predicted results, distinct clusters can be seen, one for each number of nodes, and the shapes are similar.

Still, Figure 4.1 does not give precise numbers. Therefore, we show in Table 4.2 the average and worst-case differences between our predicted and actual results for each NAS program. This shows that on average our predictions are within 10% over all possible schedules.

Examining the problem space: In our previous work [17], we showed that the behavior of a parallel program at different gear settings was dependent on program speedup and memory pressure. The better the speedup, the lower the energy premium to run using more nodes. At
Table 4.2: Average difference between predicted results and actual results (in absolute value).

<table>
<thead>
<tr>
<th>Program</th>
<th>Time</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>10%</td>
<td>5.0%</td>
</tr>
<tr>
<td>CG</td>
<td>7.8%</td>
<td>8.3%</td>
</tr>
<tr>
<td>EP</td>
<td>11%</td>
<td>4.9%</td>
</tr>
<tr>
<td>IS</td>
<td>4.6%</td>
<td>9.7%</td>
</tr>
<tr>
<td>FT</td>
<td>6.1%</td>
<td>3.6%</td>
</tr>
<tr>
<td>LU</td>
<td>9.3%</td>
<td>4.5%</td>
</tr>
<tr>
<td>MG</td>
<td>8.2%</td>
<td>5.7%</td>
</tr>
<tr>
<td>SP</td>
<td>8.5%</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

The limit, if speedup is perfect, then there is no additional energy cost because doubling the nodes halves the execution time. Also, the higher the memory pressure, the greater benefit in running at a lower gear, because the CPU spends a large amount of time waiting on memory. Together, these two factors control the behavior of a program as it relates to our algorithm and model.

To illustrate that our algorithm can support an arbitrary combination of speedup/memory pressure, we constructed a set of four benchmarks. Figure 4.2 lists the characteristics of each benchmark—note that each NAS benchmark exists somewhere in the interior of this space. In other words, these are the extremes (all four combinations of lowest/highest OPM and no/perfect speedup) of the problem space.

We created six synthetic benchmarks, each with two phases. A benchmark consisted of two (different) programs shown in Figure 4.2. We tried all unique combinations (we did not reverse the order of phases).

The results for the six benchmarks are presented in Table 4.3. There are several interesting aspects of these results. First, in three of six cases, we found the optimal schedule, and in four of the six cases, we executed the program the minimal number of times our framework
allows (11). In two of the three cases where the schedule is not optimal, the difference in time is less than 3%.

However, in one case, the difference is rather large (22%). There are two contributing factors here. First, the second phase has excellent speedup and a high OPM—meaning that using a lower gear will actually take more energy. Second, the chosen schedule is (4, [0,0]), and it turned out that the second verification test, where more nodes are tried, tested (8, [4,4]). Executing that schedule showed that the energy limit was exceeded. Our algorithm does not realize that (8, [4,3]) will actually use less energy. In fact, the optimal schedule is (8, [4,1]). We will investigate this problem in future work.

In this case, we underpredict time and energy by about 10%. In addition, the energy limit is such that candidate schedules are clustered together in the problem space. Our system believes that each is below the limit, and so ends up trying each of them iteratively. We are further investigating this.
Table 4.3: Results of running our algorithm on synthetic benchmarks. Each program is two phases. For example, (Poor/Low, Good/Low) indicates that the first phase had poor speedup and a low OPM, and the second phase had good (perfect) speedup and a low OPM.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Poor/Low, Good/Low</td>
<td>36</td>
<td>12</td>
<td>144</td>
<td>(8, 2,2)</td>
<td>2</td>
</tr>
<tr>
<td>Poor/Low, Poor/High</td>
<td>43</td>
<td>11</td>
<td>144</td>
<td>(8, 1,1)</td>
<td>1</td>
</tr>
<tr>
<td>Poor/Low, Good/High</td>
<td>30</td>
<td>11</td>
<td>144</td>
<td>(4, 0,0)</td>
<td>2</td>
</tr>
<tr>
<td>Good/Low, Poor/High</td>
<td>26</td>
<td>25</td>
<td>144</td>
<td>(4, 0,0)</td>
<td>1</td>
</tr>
<tr>
<td>Good/Low, Good/High</td>
<td>18</td>
<td>11</td>
<td>144</td>
<td>(8, 2,0)</td>
<td>1</td>
</tr>
<tr>
<td>Poor/High, Good/High</td>
<td>21</td>
<td>11</td>
<td>144</td>
<td>(4, 0,0)</td>
<td>1</td>
</tr>
</tbody>
</table>

Overall, given that these six tests represent extremes of our problem space, we believe that these results show that our system is in general effective.

Sources of error: Here we examine the sources of error in our system. First, the power meter has error. To further understand this error, we ran a program that consumes a constant amount of power for any time $T$. Then, we varied $T$, measuring the standard deviation. We found that a relatively small amount of error occurs in the consumed energy. In particular, we found in general that the largest disparity in the measured energy was 100 joules. Considering that a typical NAS program consumes thousands of joules even when running for only a handful of seconds, we attempted to make executions last at least 30 seconds. This means the error due to the meter was less than 1%.

Second, as mentioned earlier, we model idle time assuming that it is independent of gear. As idle time includes blocking communication calls, this is not strictly true. However, our measurements have shown that strictly receiving data in MPI is relatively insensitive to gear. We have not yet measured the precise error here, but our initial results suggest that it is quite small. It does increase as the gear slows, so our predictions are more accurate at faster gears.
Effect of energy limit: In our NAS programs, we chose an energy limit that was 15% less than the peak energy that each program consumed. Here, we investigate the effect of varying the energy limit. We vary the limit from 13kJ, which for this program is the minimum needed to find a valid schedule, to 25kJ, where the chosen schedule is to use the maximum number of nodes at the fastest gear.

Figure 4.3 shows the behavior for a one-phase synthetic benchmark that has poor speedup and a small OPM. The Figure shows the number of executions as well as the quality of the schedule. Note that in general, either two or three executions are needed. In addition, the quality of the schedule is typically quite good.

Two things are of note here. First, at 16.5kJ, a much greater number of executions is needed. This is due to the fact that there are a large number of executions whose energy consumption is predicted to be close to the limit. A small error in energy prediction may cause a candidate schedule to exceed the energy limit when it is believed to be within the limit (or vice versa). This causes “nearby” schedules to be evaluated.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CG, without</td>
<td>23.5</td>
<td>13</td>
<td>24</td>
<td>(2, [0])</td>
<td>6</td>
</tr>
<tr>
<td>CG, with</td>
<td>23.5</td>
<td>11</td>
<td>24</td>
<td>(4, [1])</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.4: Performance of CG with and without superlinearity filtering.

Second, at 18kJ, our system produces a schedule that has the largest difference from optimal. We observe that while more executions will always mean a better schedule is chosen, more executions does not necessarily mean that a better relative schedule (compared to that with a different limit) will be chosen. energy limit is the primary factor in CG: CG presented a potential problem for our system. This is because on our cluster, the one node program is out of core, which results in superlinear speedup for the two-, four-, and eight-node programs. The resulting effect is that the values of $F_p$ and $F_s$ are distorted, which also (originally) made our predictions potentially inaccurate. The upper row of Table 4.4 shows the results of running CG, before filtering. The key problem when not filtering is that the inaccurate predictions cause the gear/node validation steps to constantly fail because a better schedule typically arises.

To handle this, we used a simple filtering technique to remove anomalous results. Specifically, when our system detects superlinear speedup, we compute $F_p$ and $F_s$ using the first number of nodes that results in an in-core program. As can be seen from the lower row of the table, this greatly improved our results in two ways. First, we choose a better quality schedule; twice as many nodes are used at just one lower gear, which results in a 10% better running time. Second, we decreased the number of executions needed by two.
Chapter 5

Conclusion

This thesis has addressed the problem of finding a schedule that minimizes execution time on a power-scalable cluster and a maximum energy budget. Our approach was to use a combination of performance prediction and profiling, backed up by actual program execution. Results showed that typically, only a handful of iterations of a given program need to be executed to find a schedule that results in a near-optimal execution time. In particular, over all NAS benchmarks, typically less than 15 executions (of just a few iterations each) are needed to choose a schedule. Equally importantly, the quality of the chosen schedule is usually within 10% of the optimal schedule that is found by running all schedules exhaustively. Furthermore, the difference is usually much smaller.

While we are encouraged by our results, there are still open questions that we intend to address in future work. First, we hope to implement our system on a much larger cluster. This will help us determine the robustness of our system. Second, we would like to relax some of the assumptions that we have made in this thesis; in particular, we would like to expand our model to include load imbalance as well as reducible work. Finally, we plan to use the infrastructure created in this work within a general power-aware MPI runtime system.


